# Trabajo Práctico 1
# Programación Funcional

Paradigmas de Lenguajes de Programación — $2^{do}$ cuat. 2017

Fecha de entrega: 12 de septiembre

## 1. Introducción

El objetivo de este trabajo es implementar un diccionario sobre una variante del árbol 2-3, en la cual los datos se guardan en las hojas, mientras que los nodos internos se utilizan como índice. También se definirán algunas funciones sobre este tipo de árboles y, finalmente, se utilizará el diccionario para jugar a la búsqueda del tesoro.

El juego de la búsqueda del tesoro consiste en seguir pistas una por una, cada pista conduciendo a la siguiente, para encontrar - al cabo de la última pista - un tesoro escondido.

## 2. Implementación

### 2.1. Tipos de datos utilizados

Un árbol 2-3 es una estructura generalmente utilizada como diccionario en medios físicos lentos, como pueden ser discos duros, memorias USB, etc. Su construcción es simple, sus nodos internos tienen 2 o 3 hijos y guardan 1 o 2 valores respectivamente, mientras que sus hojas sólo guardan 1 valor. Cabe notar que el tipo de las hojas y de los nodos internos no deben ser necesariamente el mismo.
Al utilizarlo como diccionario, se utilizan las hojas como valores y los nodos de manera muy similar al Árbol Binario de Búsqueda. El árbol debe cumplir con el siguiente invariante:

- Todo nodo interno tiene 2 o 3 hijos.
- Todas las hojas están en el mismo nivel (perfectamente balanceado).
- Todo valor almacenado en las hojas del subárbol izquierdo tiene claves menores que la del subárbol derecho. Y análogamente para el caso de los nodos ternarios se cumple esa relación entre el subárbol izquierdo y el medio, y entre el medio y el derecho.
- En el caso de un nodo binario, la clave guardada indica la mínima clave del subárbol derecho.
- En el caso de un nodo ternario las claves guardadas indican la mínima clave de los subárboles medio y derecho respectivamente.

Es importante mencionar que, al utilizarlo como diccionario, el árbol no crece por las hojas, sino que crece por la raíz. Lo que sucede al agregar es que se insertan nodos intermedios con

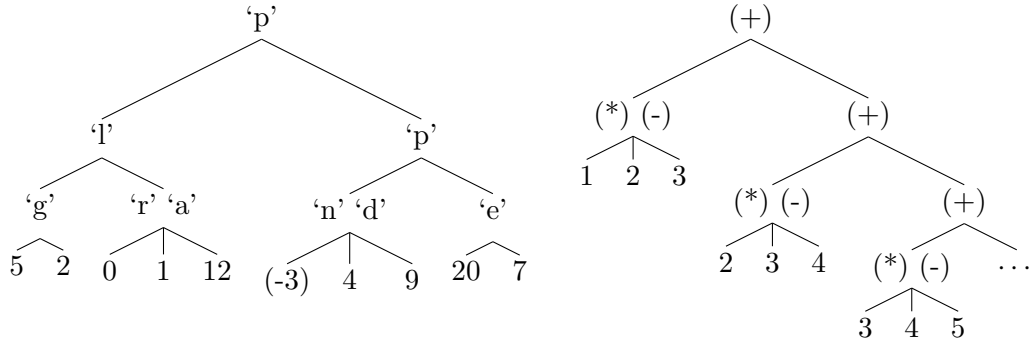
más capacidad a modo de ensanchar el árbol, propagándose el ensanchamiento hasta la raíz si es necesario.

Para más detalles, pueden buscar Árbol 2-3 en Internet[1].

## 2.2. Módulo Arbol23

En el módulo `Arbol23.hs` se ubicarán las funciones relacionadas con el manejo del árbol (este módulo no requiere que el árbol cumpla invariante alguno; es decir, permite crear árboles no balanceados y con sus nodos en cualquier orden). Se encuentra ya definida la función `show`.

Este módulo permite definir y utilizar árboles no vacíos, con un tipo para las hojas y otro para los nodos internos, donde cada nodo interno tiene dos o tres hijos. A su vez, los nodos con tres hijos son dobles (contienen dos valores). Eventualmente puede haber ramas infinitas.




Ejemplos de árboles de tipo `Arbol23 Int Char` y `Arbol23 Int (Int->Int ->Int)`. Pueden encontrar sus definiciones en el módulo Arbol23 como arbolito4 y arbolito3.

### Ejercicio 1

Dar el tipo y definir la función `foldA23` que implemente un esquema de recursión estructural para el tipo `Arbol23`. Se permite utilizar recursión explícita para definir esta función.

### Ejercicio 2

Definir las siguientes funciones, usando `case` o `foldA23` según sea conveniente:

- `hojas::Arbol23 a b->[a]`, que lista las hojas del árbol de izquierda a derecha.
- `internos::Arbol23 a b->[b]`, que lista los nodos internos en preorden (DFS).
- `esHoja::Arbol23 a b->Bool`, que indica si el árbol es una hoja.

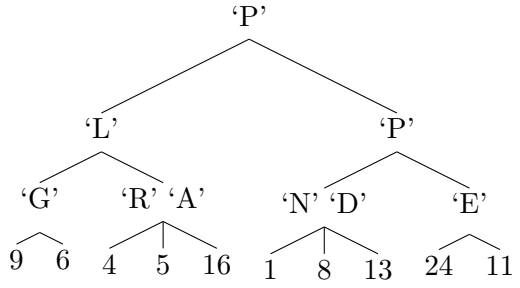
### Ejercicio 3

Definir la función `mapA23::(a->c)->(b->d)->Arbol23 a b->Arbol23 c d`.

Esta función recibe como parámetros dos funciones, y aplica la primera a todas las hojas del árbol, y la segunda a todos los nodos internos (en el caso de los nodos con tres hijos, aplica la segunda función a cada uno de sus datos).

Por ejemplo `mapA23 (+4) toUpper` aplicada al primer árbol de arriba devuelve:

[1] https://www.slideshare.net/evansbv/arboles-23-insertar-eliminar y otras fuentes.



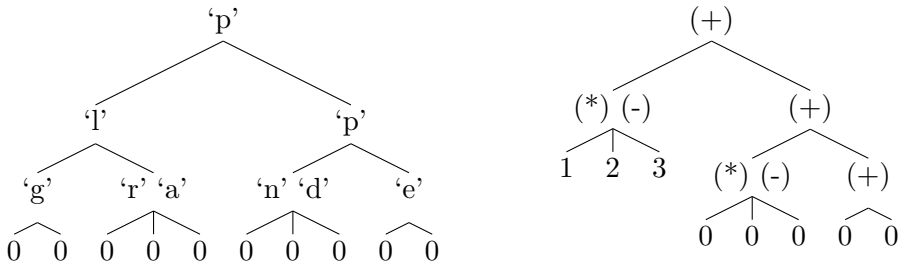

**Nota:** para probar este ejemplo es necesario importar el módulo Char.

## Ejercicio 4

Implementar la función `truncar::a->Integer->Arbol23 a b->Arbol23 a b`.

Esta función, dados un valor de tipo `a`, un entero no negativo y un árbol, devuelve un árbol que coincide con el original desde la raíz hasta el nivel indicado por el entero. Todo lo que haya por debajo de ese nivel es reemplazado por una hoja cuyo valor es el dato pasado como parámetro. Debe funcionar para árboles infinitos.

Por ejemplo, si aplicamos `truncar 0 3` a los árboles de arriba, obtenemos:




**Notas:** el resultado de truncar un árbol hasta el nivel 0 es sólo una hoja. Hasta el nivel 1, es la raíz original, cambiando los hijos por hojas. Para resolver este ejercicio es importante aprovechar los conceptos de currificación y alto orden. Sugerimos usar `foldNat` (de la práctica) para la recorrida por niveles, y que sea la función resultante quien recorra el árbol.

## Ejercicio 5

Definir la función `evaluar::Arbol23 a (a->a->a)->a`, que interpreta al árbol como el árbol sintáctico de un término y lo evalúa. Es decir, las funciones de los nodos internos se aplican a los hijos. Se asume que los nodos de tres hijos están asociados a izquierda: es decir, la primera función se aplica a los primeros dos hijos, y la segunda al resultado de esto con el tercero.

Sea `t` el árbol infinito del ejemplo:

`evaluar (truncar 0 3 t)` $\rightsquigarrow$ `-1`
`evaluar (truncar 0 4 t)` $\rightsquigarrow$ `1`
`evaluar (truncar 0 5 t)` $\rightsquigarrow$ `8`
`evaluar (truncar 0 6 t)` $\rightsquigarrow$ `22`

## 2.3. Módulo `Diccionario`

Se introducen los siguientes tipos para representar algunos conceptos:

```
type Comp clave = clave->clave->Bool
type Estr clave valor = Arbol23 (clave,valor) clave
```

El tipo `Comp` representa un comparador de claves (no exigimos que las claves sean comparables por $<$, pero sí que el usuario provea una función que las compare). El tipo `Estr` es la estructura sobre la cual se construirá el diccionario. Para los diccionarios vacíos, la estructura será `Nothing`. Para los demás, ésta contendrá un Arbol23 con las siguientes características: en cada hoja se guardará una clave con su valor correspondiente, mientras que los nodos internos servirán como índice para buscar las claves.

**Condiciones de uso:** se espera que la función de comparación sea estricta (que devuelva `False` si las claves son iguales). Además, no se definirán claves repetidas.

La definición del tipo diccionario es la siguiente:

```
data Diccionario clave valor =
Dicc {cmp :: Comp clave, estructura :: Maybe (Estr clave valor)}
```

Esto es lo mismo que decir:

`data Diccionario clave valor = Dicc (Comp clave) (Maybe (Estr clave valor))`,

pero se definen automáticamente los proyectores `cmp` y `estructura`. Es decir, si tenemos un diccionario llamado `d`, la expresión `cmp d` nos da el comparador del diccionario `d`, y `estructura d` su estructura (o `Nothing` en el caso del diccionario vacío).

Las funciones que deberán definir en este módulo son las siguientes:

### Ejercicio 6

`vacío::Comp clave->Diccionario clave valor`, que crea un diccionario vacío con la función de comparación pasada como parámetro.

### Ejercicio 7

`definir::clave->valor->Diccionario clave valor->Diccionario clave valor` que define una clave con su valor correspondiente en el diccionario. La clave debe insertarse utilizando el comparador propio del diccionario. Ya se encuentra definida la función `insertar::clave->valor->Comp clave->Estr clave valor->Estr clave valor`, que se encarga de insertar un par `(clave,valor)` en un Arbol23 manteniendo el balanceo; deberán llamarla en el contexto adecuado y con los argumentos que correspondan. Se asume que no se definirán claves repetidas.

### Ejercicio 8

`obtener::Eq clave⇒clave->Diccionario clave valor->Maybe valor`, que busca una clave en el árbol y devuelve el valor correspondiente, o `Nothing` si la clave no se encuentra definida. La búsqueda *debe* ser eficiente (no debe recorrer más de una rama del árbol).

**Sugerencia:** definir una función auxiliar para buscar la clave en el árbol.

### Ejercicio 9

`claves::Diccionario clave valor -> [clave]`, que devuelve una lista con todas las claves del diccionario.

### Ejercicio 10

`búsquedaDelTesoro::Eq a => a -> (a -> Bool) -> Diccionario a a -> Maybe a`

La búsqueda del tesoro sobre un diccionario se juega de la siguiente manera: se cuenta con una pista inicial y una función que, dado un valor, permite distinguir si es o no el tesoro. Se busca la primera pista en el diccionario, y se utiliza el valor encontrado como siguiente pista, y así sucesivamente hasta encontrar el tesoro, el cual será el resultado de la función. Si en algún momento se llega a una pista que no está definida, se deberá devolver `Nothing`.

**Sugerencia:** usar las funciones `iterate`, `>>=` y algún esquema de recursión que permita recorrer listas infinitas.

## Pautas de Entrega

Se debe entregar el código impreso con la implementación de las funciones pedidas. Cada función asociada a los ejercicios debe contar con ejemplos que muestren que exhibe la funcionalidad solicitada. Además, se debe enviar un e-mail conteniendo el código fuente en Haskell a la dirección plp-docentes@dc.uba.ar. Dicho mail debe cumplir con el siguiente formato:

- El título debe ser `[PLP;TP-PF]` seguido inmediatamente del nombre del grupo.
- El código Haskell debe acompañar el e-mail y lo debe hacer en forma de archivo adjunto (puede adjuntarse un `.zip` o `.tar.gz`).
- El código entregado **debe** incluir tests que permitan probar las funciones definidas.

El código debe poder ser ejecutado en Haskell2010. No es necesario entregar un informe sobre el trabajo, alcanza con que el código esté **adecuadamente** comentado (son comentarios adecuados los que ayudan a entender lo que no es evidente o explican decisiones tomadas; no son adecuadas las traducciones al castellano del código). Los objetivos a evaluar son:

- Corrección.
- Declaratividad.
- Prolijidad: evitar repetir código innecesariamente y usar adecuadamente las funciones previamente definidas (tener en cuenta tanto las funciones definidas en el enunciado como las definidas por ustedes mismos).
- Uso adecuado de funciones de alto orden, currificación y esquemas de recursión: Es necesario para los ejercicios que usen las funciones que vimos en clase y aprovecharlas, por ejemplo, usar zip, map, filter, take, takeWhile, dropWhile, foldr, foldl, listas por comprensión, etc, cuando sea necesario y no volver a implementarlas.

Salvo donde se indique lo contrario, **no se permite utilizar recursión explícita**, dado que la idea del TP es aprender a aprovechar las características enumeradas en el ítem anterior.

Se permite utilizar listas por comprensión y esquemas de recursión definidos en el preludio de Haskell y los módulos `Prelude`, `List`, `Maybe`, `Data.Char`, `Data.Function`, `Data.List`, `Data.Maybe`, `Data.Ord` y `Data.Tuple`. Las sugerencias de los ejercicios pueden ayudar, pero no es obligatorio seguirlas. Pueden escribirse todas las funciones auxiliares que se requieran, pero estas no pueden usar recursión explícita (ni mutua, ni simulada con `fix`).

**Importante:** se admitirá un único envío, sin excepción alguna. Por favor planifiquen el trabajo para llegar a tiempo con la entrega.

**Tests:** se recomienda la codificación de tests. Tanto HUnit https://hackage.haskell.org/package/HUnit como HSpec https://hackage.haskell.org/package/hspec permiten hacerlo con facilidad.

Para instalar HUnit usar: `> cabal install hunit`

Para instalar cabal ver: https://wiki.haskell.org/Cabal-Install

## Referencias del lenguaje Haskell

Como principales referencias del lenguaje de programación Haskell, mencionaremos:

- **The Haskell 2010 Language Report:** el reporte oficial de la última versión del lenguaje Haskell a la fecha, disponible online en: http://www.haskell.org/onlinereport/haskell2010.
- **Learn You a Haskell for Great Good!**: libro accesible, para todas las edades, cubriendo todos los aspectos del lenguaje, notoriamente ilustrado, disponible online en http://learnyouahaskell.com/chapters.
- **Real World Haskell**: libro apuntado a zanjar la brecha de aplicación de Haskell, enfocándose principalmente en la utilización de estructuras de datos funcionales en la "vida real", disponible online en http://book.realworldhaskell.org/read.
- **Hoogle**: buscador que acepta tanto nombres de funciones y módulos, como signaturas y tipos *parciales*, online en http://www.haskell.org/hoogle.
- **Hayoo!**: buscador de módulos no estándar (i.e. aquellos no necesariamente incluídos con la plataforma Haskell, sino a través de **Hackage**), online en http://holumbus.fh-wedel.de/hayoo/hayoo.html.